Programowanie Komputerów 4

Projekt „*Eurobiznes*”

17/06/2022

Autor: Jakub Hoś

Prowadzący: prof. Roman Starosolski

Temat projektu

Projekt zakłada stworzenie aplikacji będącej sieciową wersją popularnej gry planszowej „Eurobiznes”. Gracze po dołączeniu do gry otrzymują pewną wartość „$”(domyślnie 3000$), czyli głównej waluty w grze. Gra trwa do momentu, aż wszyscy oprócz jednego gracza zbankrutują, bądź do określonego czasu(domyślny czas rozgrywki to 20 minut).

Analiza tematu

Doprecyzowanie tematu

Aplikacja pozwala na rozgrywkę online dla 2,3 lub 4 osób. Gra działa w sieci LAN oraz sieci WAN(jeżeli któryś z graczy dysponuje publicznym adresem IP). Połączenie jest realizowane na porcie 53000 przy pomocy niezawodnego protokołu TCP. Po dołączeniu wszystkich graczy do rozgrywki rozpoczyna się gra. Oprócz graficznej wizualizacji aktualnego stanu rozgrywki gra posiada ścieżkę dźwiękową zależną od aktualnej sytuacji na planszy.

Uzasadnienie wyboru klas

Krótki opis utworzonych klas:

- GUI – odpowiada za wyświetlanie interfejsu graficznego oraz pobieranie danych od użytkownika(m.in. nazwę gracza, kolor gracza, adres IP itp.)
- button – klasa realizująca funkcjonalność przycisku. Do przycisku można wpisywać tekst (input button) lub go wybrać(zwykły przycisk)– zależy to od tego, jakiego użyjemy konstruktora
- game – klasa odpowiadająca za wyświetlanie aktualnego stanu gry, obsługę akcji po rzucie kostką oraz komunikację sieciową
- player – klasa przechowująca informacje na temat gracza oraz teksturę jego pionka
- field – klasa abstrakcyjna, bazowa dla klas: city, industry, jail, fastTravell, chance, special, które odpowiadają za akcje wykonywane po wejściu gracze na dane pole oraz wyświetlanie zmian

Algorytmy

W grze zastosowano prosty algorytm, używany w przypadku, gdy graczowi zabraknie środków na zapłatę czynszu, kary bądź podatku, a wartość jego majątku jest w stanie pokryć dług.

Sposób działania:

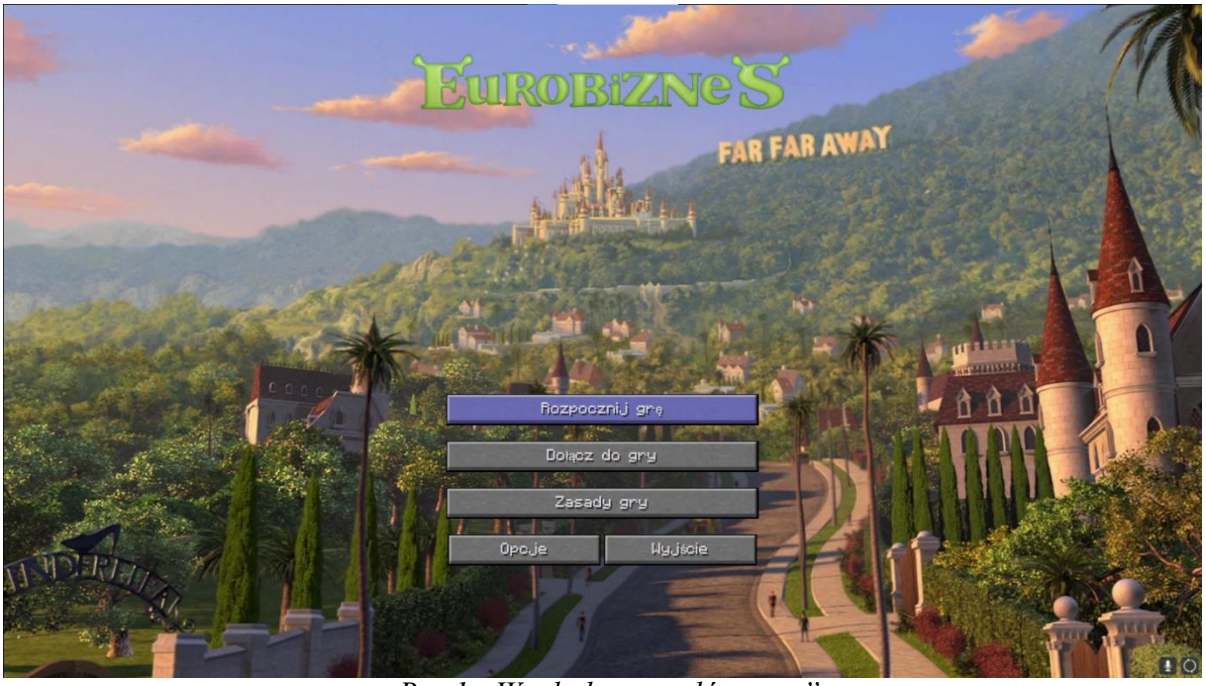
- wszystkie posiadłości gracza są umieszczane w wektorze jako para liczb całkowitych – id pola oraz jego wartość
- wektor jest sortowany rosnąco według wartości pól
- pola są sprzedawane tak długo, aż dług nie zostanie pokryty
- pozostałe środki ze sprzedaży(jeżeli takowe istnieją) zostają dodane do stanu konta gracza

Do generowania liczb losowych został użyty generator *mt19937*.

Biblioteki
W programie wykorzystano bibliotekę zewnętrzną SFML w wersji 2.5

Specyfikacja zewnętrzna
Nawigacja w grze odbywa się za pomocą klawiatury. Opcje w menu wybiera się za pomocą „strzałek”. Do podejmowania decyzji należy używać klawisz Enter, „T” oraz „N”. Okno można zamknąć używając myszki i przycisku X(na pasku menu w górnej części ekranu).

Po uruchomieniu gry użytkownik posiada do dyspozycji menu.

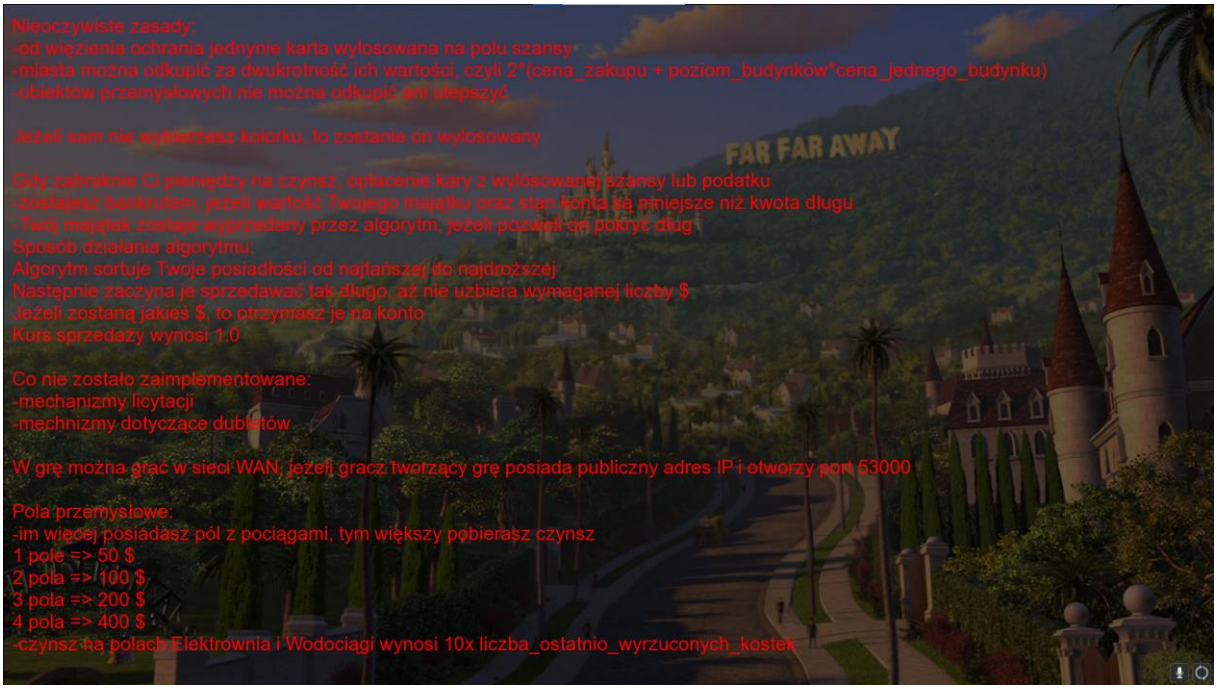


Rys.1 „Wygląd menu głównego”

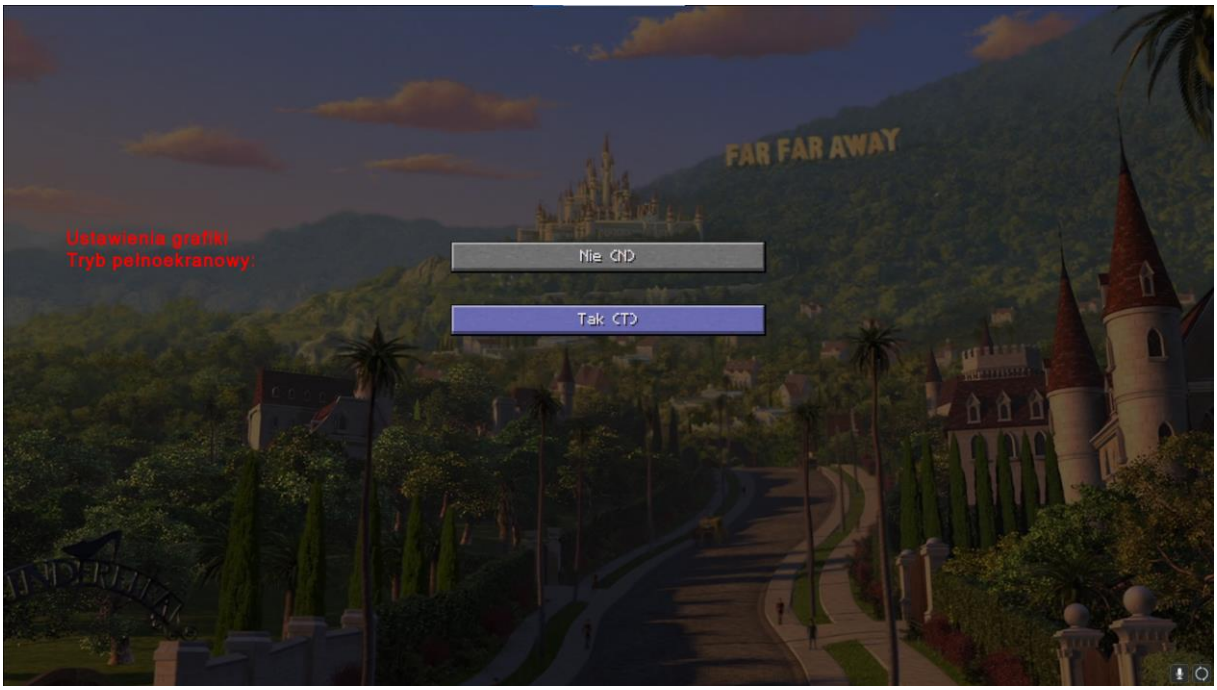
Po wybraniu którejś z opcji użytkownik ma do dyspozycji jedno z podmenu.

Rys.2 „Wygląd podmenu rozpoczęcia gry”

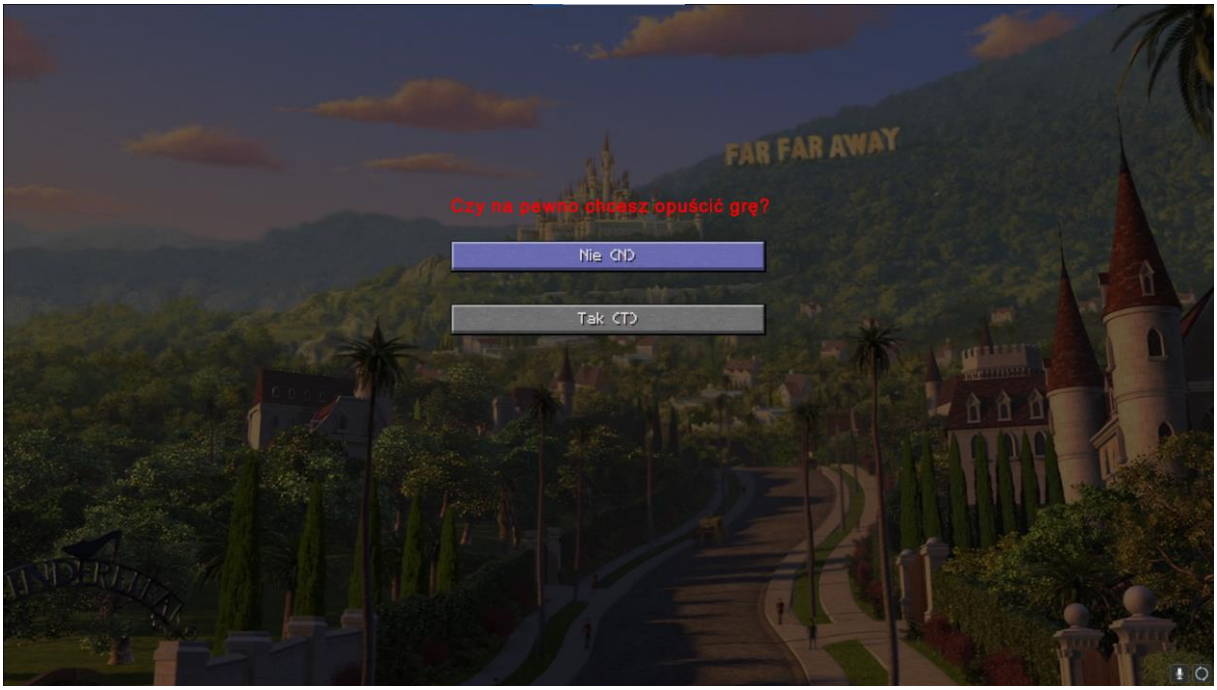
Rys.3 „Wygląd podmenu dołączania do gry”



Rys.4 „Wygląd podmenu z zasadami gry”

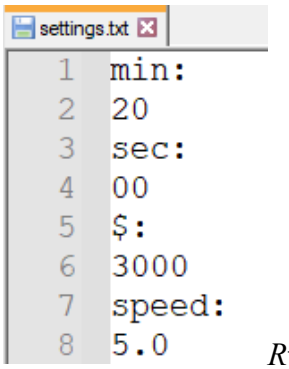


Rys.5 „Podmenu z dostępnymi opcjami”



Rys.6 „Podmenu opuszczania gry”

Dodatkowe opcje rozgrywki(czas trwania gry, początkowa wartość konta, szybkość poruszania się pionków) można ustawić w pliku settings.txt, który znajduję się w folderze settings.



Rys.7 „Dostępne opcje w pliku settings/settings.txt”

Wszelkie nieoczywiste i/lub zmienione zasady gry są opisane w podmenu „Zasady gry”.

Akcje związane z poszczególnymi klasami pól:

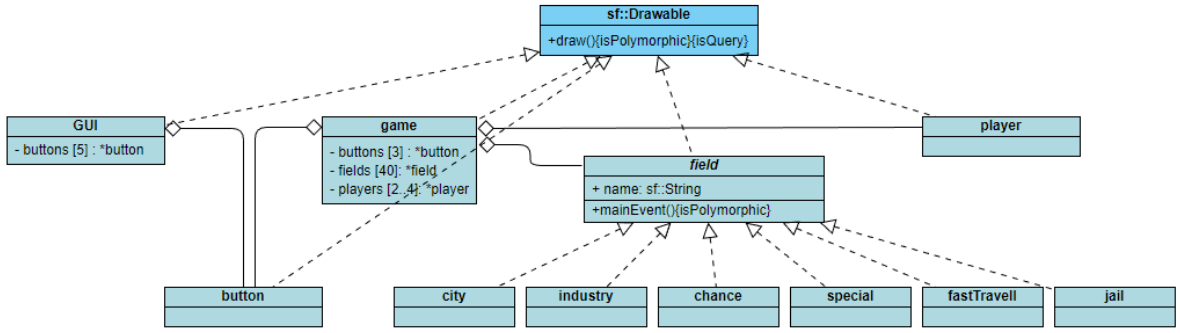
- **Miasto**: Gracz w zależności od tego, do kogo należy miasto po wejściu na pole może je kupić, odkupić(za dwukrotność jego wartości) lub ulepszyć. Jeżeli miasto nie należy do niego, to musi on zapłacić czynsz.
- **Pola przemysłowe**: Działa podobnie jak miasto, ale nie można go odkupić ani ulepszyć. Wysokość czynszu zależy od ilości posiadanych pół tego typu lub ostatnio wyrzuconych oczek.
- **Więzienie:** Blokuje gracza na 2 kolejki, a uchronić przed nim może jedynie szczególna karta pociągnięta z pola szansy.
- **Pole specjalne:** Do konta gracza dodawany/odejmowany jest bonus pola.
- **Pole „szybkiej podróży”**: Przenosi gracza do więzienia, chyba że ten posiada specjalną kartę chroniącą przed tą karą.
- **Szansa**: Uruchamiana jest akcja – np. zmiana stanu konta, powrót na dane pole, zdobycie karty chroniącej przed więzieniem, która dotyczy gracza, który stanął na polu szansy.

Oprócz graficznej wizualizacji stanu rozgrywki, wyświetlane są komunikaty o decyzjach podjętych przez graczy(na 5 sekund).

Rys.8 „Wygląd rozgrywki”

Specyfikacja wewnętrzna

Diagram klas



Powiązania pomiędzy klasami

Wszystkie klasy dziedziczą interfejs umożliwiający wyświetlanie z klasy *sf::Drawable*.

Wszystkie agregacje polegają na przechowywaniu wektorów lub tablic std::shared_ptr lub std::unique_ptr na obiekty innych klas.

Obiekt typu GUI pobiera od gracza informacje, a następnie nasłuchuje port 53000/łączy się do podanego IP.

Podczas rozgrywki obiekt typu game uruchamia metodę mainEvent() odpowiedniego pola, która dokonuje zmian w własnych polach i/lub obiektach graczy.

Pionki „przesuwa” jedna z metod klasy game.

Z klasy field dziedziczy 6 klas, które istotnie się od siebie różnią, dlatego zostały wydzielone.

Po zakończeniu rozgrywki obiekt typu game jest niszczony, a razem z nim wszystkie agreganty. Gracz wraca do menu.

Istotne metody klas

- sf::Drawable
  - `virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const` – wyświetlenie obiektu na ekranie
- button:
  - `void select()` – wybranie przycisku
  - `void unselect()` – wybranie innego przycisku
  - `void addLetter(char x)` – wprowadzenie litery
  - `void clearInnerText()` – wyczyszczenie zwartości pola
  - `void deleteLetter()` – usunięcie ostatniej litery
- field oraz klasy pochodne:
  - `virtual void mainEventServer(std::vector<std::shared_ptr<player>> gg, int i, bool dec, sf::Text& text) = 0` – główny event pola, jako parametr przyjmuje wektor wskaźników na graczy, id

aktualnego gracza, bit mówiący o decyzji gracza oraz referencje do obiektu typu sf::Text, który ma edytować
    - `virtual bool showButton(std::vector<std::shared_ptr<player>> gg, int i, sf::Text &text) = 0` – zwraca informację, czy wyświetlić przyciski do podjęcia decyzji oraz edytuje tekst odnośnie podjęcia decyzji
    - `virtual int getOwner() = 0` – zwraca id właściciela pola
    - `virtual int getValue() = 0` – zwraca wartość pola
    - `virtual void sellField() = 0` – sprzedaje pole
    - `virtual sf::Vector2f getPlace(int id)` – zwraca wektor współrzędnych, na jakie ma udać się gracz
- GUI:
    - `void start()` – tworzy okno, rozpoczyna wyświetlanie menu
    - `bool validIp(std::string ip)` – sprawdza poprawność wprowadzonego przez gracza adresu IP
- game:
    - `void readSettings()` – odczytuje ustawienia z pliku .txt
    - `void startTimer()` – uruchamia odliczanie czasu gry
    - `void sellProperties()` – windykacja posiadłości aktualnego gracza
    - `void loadBoard()` – tworzy obiekty klasy field
    - `void movePlayer()` – przesuwa pionek gracza na nowe miejsce
    - `void setLabels()` – ustawia właściwości tabeli z graczami i ich stanami kont
    - `void loadTexts()` – ustawia właściwości obiektów typu sf::Text
    - `void loadTextures()` – wczytuje tekstury z plików
    - `void connectWithPlayer(std::string serverNick, int serverColor)` – nawiązuje połączenie z graczem
    - `void connectWithSerwer(std::string nick, int c, sf::IpAddress serwerIP, int port = 53000)` – nawiązuje połącznie z serwerem
    - `void sendPlayers()` – wysyła informacje na temat graczy
    - `void getPlayers()` – odbiera informacje na temat graczy
    - `void setEndTextAndSound()` – generuję tablicę wyników i zmienia odtwarzaną melodię(zależenie od wygranej lub przegranej gracza)
    - `void loadSound()` – wczytuje dźwięki z plików
    - `void removePlayer()` – usuwa wszystkie posiadłości gracza, gdy ten zbankrutował
    - `void hideInfo(int interwal)` – metoda ukrywa informacje po zadanym czasie, który przyjmuje jako parametr
    - `void recive()` – metoda odbiera pakiet
    - `void send()` – metoda wysyła pakiet
- player:
    - `std::string getNick()` – metoda zwraca nick gracza

- `sf::Uint32 getColorUint32() – metoda zwraca id koloru gracza jako Uint32, aby mogło zostać łatwo przesłane przez sieć`
- `void setColor(int c) – metoda ustawia kolor gracza i przypisuje mu odpowiedni pionek`
- `void setNick(std::string n) – metoda ustawia nick gracza`
- `void setXY(float xx, float yy) – metoda ustawia pionek gracza w wybranym miejscu`

Struktury danych

Wykorzystane struktury danych to tablice, std::vector oraz std::pair.

Techniki obiektowe

W programie wykorzystano m.in. dziedziczenie wielokrotne oraz polimorfizm.

Techniki przerabiane na zajęciach

W aplikacji wykorzystano następujące techniki poznane na zajęcia:

- **regex** używany do sprawdzenia poprawności wpisanego adresu IPv4
- **ranges** używane przy sortowaniu wektorów
- **jthread** używany do równoległego obsługiwania komunikacji sieciowej, mierzenia czasu oraz wyświetlania stanu gry
- **atomic** używane dla pól, które mogą być wykorzystywane przez różne wątki w tym samym czasie
- **filesystem** używany przy odczycie ustawień z pliku

Ogólny schemat działania

Po tym, jak wszyscy klienci dołączą, serwer rozsyła pakiet informacji o wszystkich graczach. Następnie uruchamiane są wątki potrzebne do komunikacji sieciowej, mierzenia czasu gry oraz metody wyłączającej wyświetlanie informacji po 5 sekundach.

Kolejno uruchamiana jest główna pętla gry. Po zakończeniu gry wyświetlana jest tabela wyników. Po naciśnięciu dowolnego klawisza gracz zostanie przeniesiony do menu.

Testowanie i uruchamianie

- Początkowo wykorzystywałem do komunikacji sieciowej niskopoziomowe podejście z użyciem funkcji **recv**. Późniejsze wykorzystanie funkcjonalności sieciowych z biblioteki SFML(w miejsce recv) przyniosło stabilniejsze działanie gry oraz czytelniejszy kod.
- Poruszanie gracz po planszy wymagało dużego nakładu pracy i testów. Dopiero stworzenie metody przesuwającej „od zera”, tak aby pionek poruszał się tylko po jednej osi naraz rozwiązało problem.
- Gra zawieszała się za każdym razem, gdy klient oczekiwał na pakiet. Problem rozwiązało przeniesienie komunikacji sieciowej do innych wątków.
- Zrezygnowałem z używania plików **.ixx** z powodu gigantycznej liczby błędów, które nie miały miejsca przy użyciu plików .h oraz .cpp.
- Stworzenie aplikacji najpierw w wersji tekstowej, a później graficznej okazało się stratą czasu.

- Gdybym miał tworzyć tę grę jeszcze raz, podzieliłbym klasę *game* na parę mniejszych, ponieważ aktualnie realizuje ona zbyt wiele funkcjonalności naraz. Rozważyłbym również skorzystanie z wzorca projektowego.

Wnioski

Jestem zadowolony z końcowego stanu gry, z nielicznymi wyjątkami. Podczas pracy nad grą zdobyłem trochę doświadczenia w tworzeniu aplikacji z interfejsem graficznym oraz komunikacją sieciową w C++ - prędzej te dwa zagadnienia wydawały mi się być wielką filozofią.